COMP4142 E-Payment and Cryptocurrency

Group 03

Blockchain Attendance Record System

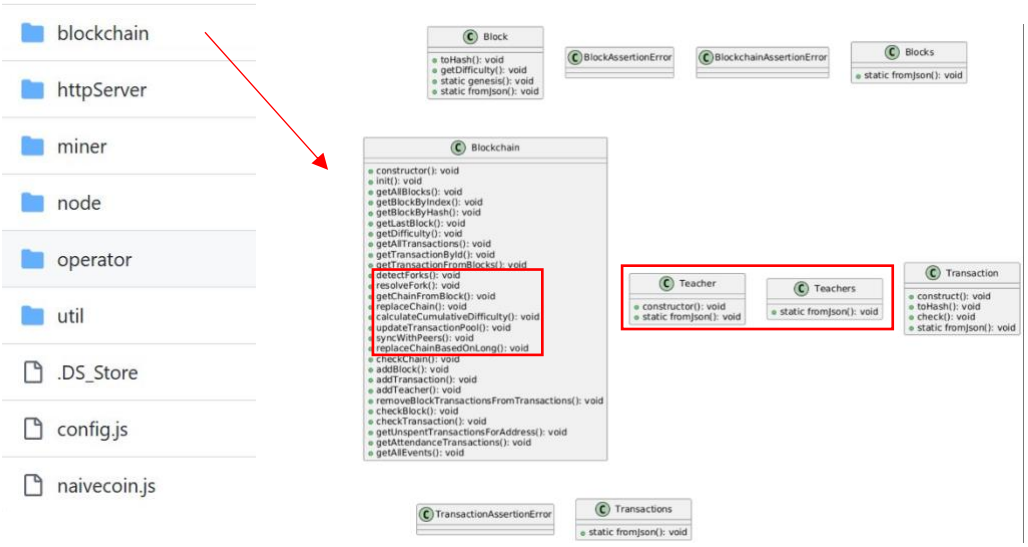
Student Name	Student ID	Responsible Part
CHEN Ziyang	21095751d	All frontend content of the Student Attendance Application, and create a user-friendly UI. Create the function "Get Secret Key" on the backend. Assist the backend to complete the work.
HE Rong	21101622d	Assist in enhancing functionality and organizing meetings. Overall flow check.
LI Shuhang	21102658d	Implementation the backend of the Student Attendance Application: Student information registration and Mint. Assist the frontend to complete the corresponding work.
LU Zhoudao	21099695d	Enhanced functionality: Achieve dynamic difficulty and Basic Fork Resolution (cumulative difficulty).
LUO Yi	21108269d	Enhanced functionality: Basic Fork Resolution.
YE Chenwei	21103853d	Implementation the backend of the Student Attendance Application: Attendance information recording and Record querying. Assist the frontend to complete the corresponding work.

Table of Content

1. System architecture overview

Our architecture is based on naivecoin (https://github.com/conradoqg/naivecoin).

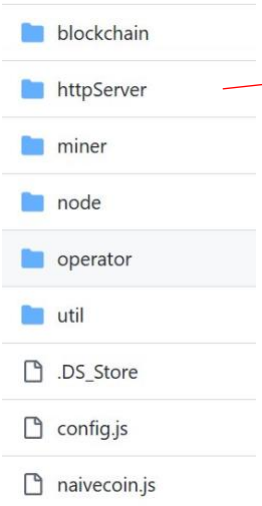
Blockchain



For 'blockchain' package, we created some specific classes for teachers to distinguish them from students; to achieve record querying, we also added some functions like 'getAttendanceTransactions()' and 'getAllEvents()' to query the data from database.

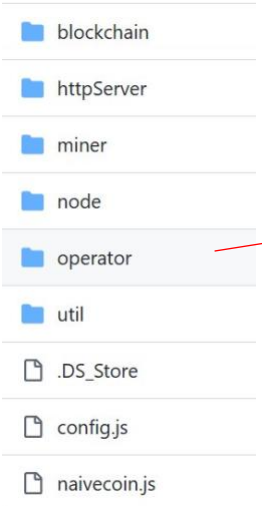
We also achieved fork resolution here; our solution is based on cumulative difficulty, but programmers can manually adjust it to choose the longest chain by functions 'replaceChainBasedOnLong()'.

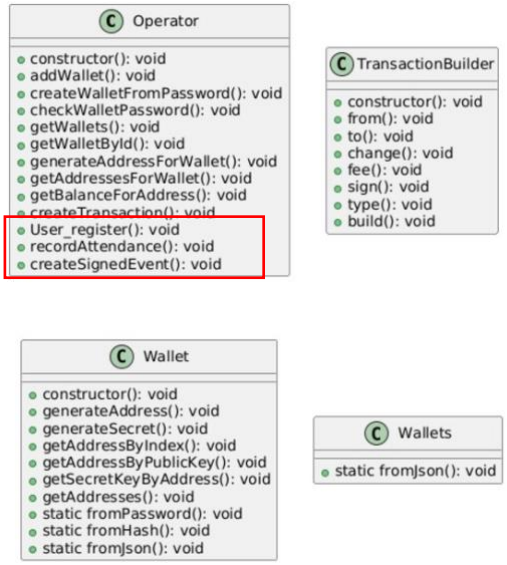
httpServer



To interface with the front-end, we also adjusted the internal functions of the HttpServer

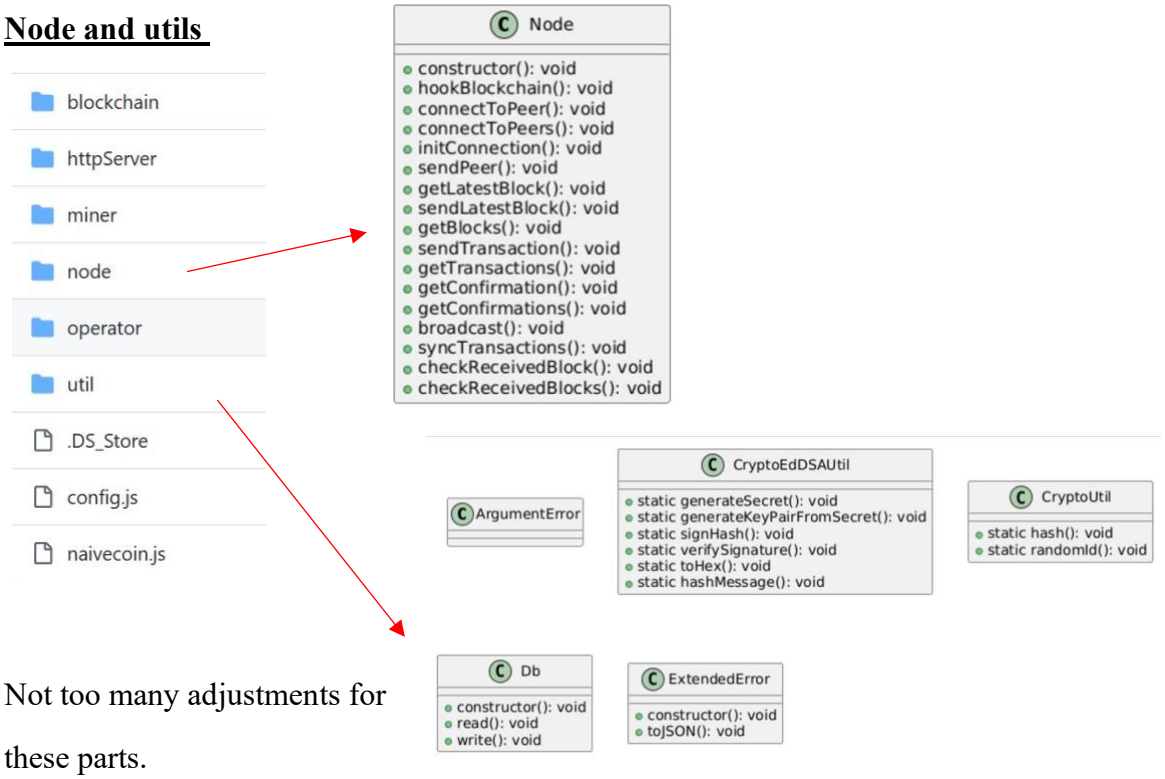
Operator





Operator is the main part of our work. We have implemented users' registrations and attendances feature here. Teachers can also use their private keys to create events and set deadlines.

Node and utils



Not too many adjustments for these parts.

We implemented dynamic difficulty in ‘config.js’.

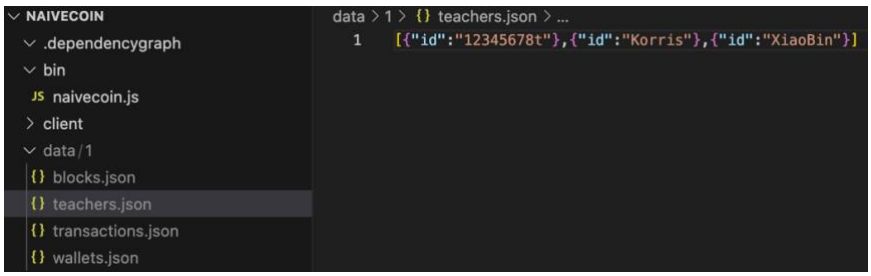
2. Key Functions Implementation

2.1 User Registration (Students and Teachers)

1. Create an object called ‘Teacher’. When initializing the code, a JSON file called ‘teachers’ will be created, and three teachers will be added into the file.

```
const R = require('ramda');
const CryptoUtil = require('../util/cryptoUtil');

class Teacher {
    constructor(id) {
        this.id = id;
    }

    static fromJson(data) {
        if (typeof data === 'object' && 'id' in data) {
            return new Teacher(data.id);
        }
        throw new Error('Invalid teacher data format');
    }
}

module.exports = Teacher;
```

```
init() {
    // Create the genesis block if the blockchain is empty
    if (this.blocks.length == 0) {
        console.info('Blockchain empty, adding genesis block');
        this.blocks.push(Block.genesis);
        this.blocksDb.write(this.blocks);
    }
    if (this.teachers.length == 0) {
        const teacher1 = new Teacher('12345678t');
        this.addTeacher(teacher1);
        const teacher2 = new Teacher('Korris');
        this.addTeacher(teacher2);
        const teacher3 = new Teacher('XiaoBin');
        this.addTeacher(teacher3);
    }
```



2. Call ‘createWalletFromPassword’ function to create a wallet for the user.

3. Construct a transaction with ‘registration’ type which includes User_ID and public key.

4. Add this transaction to the unconfirmed transaction pool, waiting for confirming.

```
async User_register(User_ID, password) {
    const newWallet = this.createWalletFromPassword(password);
    const publicKeyAddress = this.generateAddressForWallet(newWallet.id);
    const registrationTransaction = {
        id: `reg_${User_ID}`,
        type: 'registration',
        data: {
            inputs: [],
            outputs: [
                {
                    User_ID: User_ID,
                    publicKey: publicKeyAddress,
                    amount: 0,
                    address: NaN
                }
            ]
        }
    };

    await this.blockchain.addTransaction(registrationTransaction);
    console.log("newWallet:", newWallet);
    console.log("publicKeyAddress:", publicKeyAddress);
    console.log(`Student ${User_ID} registered with wallet ID: ${newWallet.id} and public address: ${publicKeyAddress}`);

    return {
        User_ID: User_ID,
        walletID: newWallet.id,
        publicKey: publicKeyAddress
    };
}
```

2.2 Create Event_ID (Only teachers can create it)

1. Check whether the user is teacher. Search whether the user is in the ‘teachers’ JSON file by using User_ID. If the user is not a teacher, throw an error.

```
async createSignedEvent(User_ID, eventID, privateKey, ddl) {
    console.log("Teachers array:", this.blockchain.teachers);
    const isTeacher = this.blockchain.teachers.some(teacher => teacher.id === User_ID);
    console.log(isTeacher);
    if (!isTeacher) {
        throw new Error(`User ${User_ID} is not a teacher and cannot create events.`);
    }
```

2. Create a deadline for this eventID. Students can only take attendance before the deadline. Transfer the user’s input into timestamp for comparing. And create the signature.

```
const parsedDate = new Date(ddl);
const timestamp = parsedDate.getTime();
const messageHash = CryptoEdDSAUtil.hashMessage(`${eventID}_${timestamp}`);
const signature = CryptoEdDSAUtil.signHash(privateKey, messageHash);
```

3. construct the transaction with ‘event’ type which includes User_ID, eventID, timestamp and signature.

4. Add this transaction to the unconfirmed transaction pool, waiting for confirming.

```
const eventTransaction = {
    id: `event_${eventID}`,
    type: 'event',
    data: {
        inputs: [],
        outputs: [
            {
                User_ID: User_ID,
                eventID: eventID,
                timestamp: timestamp,
                signature: signature,
                amount: 0,
                address: NaN
            }
        ]
    }
};

await this.blockchain.addTransaction(eventTransaction);
console.log(`Event created and signed by teacher ${User_ID} with ddl: ${ddl}`);
return { transactionId: eventTransaction.id, timestamp };
```

3. Mint unconfirmed transactions

3.1 Check the types of unconfirmed transactions. If the type of the transaction is ‘event’ or ‘attendance’, the transaction should be verified whether is valid. For other types, the transaction will be directly selected.

```
if (R.all(R.equals(false), transactionInputFoundAnywhere)) {
    if (transaction.type === 'event' || transaction.type === 'attendance') {
        const isValid = Miner.verifyEventTransaction(transaction, blockchain);
        console.log(isValid);
        if (isValid) {
            selectedTransactions.push(transaction);
        } else {
            rejectedTransactions.push(transaction);
        }
    } else if (transaction.type === 'registration') {
        selectedTransactions.push(transaction);
    } else if (transaction.type === 'regular' && negativeOutputsFound === 0) {
        selectedTransactions.push(transaction);
    } else if (transaction.type === 'reward') {
        selectedTransactions.push(transaction);
    } else if (negativeOutputsFound > 0) {
        rejectedTransactions.push(transaction);
    }
} else {
    rejectedTransactions.push(transaction);
}
candidateTransactions);
```

3.2 Verify if the transaction is valid or not (For ‘attendance’). Firstly, get the deadline from blockchain by using eventID. Compare the time when the attendance be created with the deadline in that ‘event’ transaction. Only the attendances created before deadline are valid.

3.3 Verify if the deadline in the transaction with ‘event’ type is after the current time (the deadline cannot before current time, otherwise no one can take attendance. The current time means the time when teacher create the eventID).

```
const userid = transaction.data.outputs[0].User_ID;
let event_ddl = null;
const blocks = blockchain.getAllBlocks();
if (transaction.type === 'attendance'){
    const eventid = transaction.data.outputs[0].eventID;
    for (const block of blocks) {
        for (const t of block.transactions) {
            if (t.type === 'event') {
                //console.log(t.output.timestamp);
                const output = t.data.outputs.find(output => output.eventID === eventid);
                if (output) {
                    event_ddl = output.timestamp;
                    break;
                }
            }
        }
        if (event_ddl) break;
    }
    console.error(event_ddl);
    console.error(transaction.data.outputs[0].timestamp);
    if (parseInt(event_ddl,10) < parseInt(transaction.data.outputs[0].timestamp,10)){
        console.warn(`Transaction ${transaction.id} rejected: expired.`);
        console.error(event_ddl);
    return false;
    }
    return true;
}

if (currentTime > parseInt(transaction.data.outputs[0].timestamp,10)) {
    console.warn(`Transaction ${transaction.id} rejected: expired.`);
    return false;
}
const eventHash = CryptoEdDSAUtil.hashMessage(
    `${transaction.data.outputs[0].eventID}_${transaction.data.outputs[0].timestamp}`
);
```

3.4 For both ‘event’ and ‘attendance’ transactions, get the public key by using

User_ID from the blockchain, and to verify the transactions.

```
if (currentTime > parseInt(transaction.data.outputs[0].timestamp,10)) {
    console.warn(`Transaction ${transaction.id} rejected: expired.`);
    return false;
}
const eventHash = CryptoEdDSAUtil.hashMessage(
    `${transaction.data.outputs[0].eventID}_${transaction.data.outputs[0].timestamp}`
);
let userPublicKey = null;
for (const block of blocks) {
    for (const t of block.transactions) {
        if (t.type === 'registration') {
            const output = t.data.outputs.find(output => output.User_ID === userid);
            if (output) {
                userPublicKey = output.publicKey;
                break;
            }
        }
    }
    if (userPublicKey) break;
}
console.log(userPublicKey);
const isSignatureValid = CryptoEdDSAUtil.verifySignature(
    userPublicKey,
    transaction.data.outputs[0].signature,
    eventHash
);
console.log(transaction.data.outputs[0].signature);

if (!isSignatureValid) {
    console.warn(`Transaction ${transaction.id} rejected: signature verification failed.`);
    return false;
}
return true;
```

3.5 Generate a new block to store the transactions, the transactions should be valid. And the miner will receive the reward. When the transactions are store in the blockchain, a new transaction with ‘reward’ type will also be stored in the block.

```
static generateNextBlock(rewardAddress, feeAddress, blockchain) {
    const previousBlock = blockchain.getLastBlock();
    const index = previousBlock.index + 1;
    const previousHash = previousBlock.hash;
    const timestamp = new Date().getTime() / 1000;
    const blocks = blockchain.getAllBlocks();
    const candidateTransactions = blockchain.transactions;
    const transactionsInBlocks = R.flatten(R.map(R.prop('transactions'), blocks));
    const inputTransactionsInTransaction = R.compose(R.flatten, R.map(R.compose(R.prop('inputs'), R.prop('data'))));

    let rejectedTransactions = [];
    let selectedTransactions = [];
    R.forEach((transaction) => {
        let negativeOutputsFound = 0;
        let i = 0;
        let outputsLen = transaction.data.outputs.length;

        for (i = 0; i < outputsLen; i++) {
            if (transaction.data.outputs[i].amount < 0) {
                negativeOutputsFound++;
            }
        }
        let transactionInputFoundAnywhere = R.map((input) => {
            let findInputTransactionInTransactionList = R.find(
                R.whereEq({
                    'transaction': input.transaction,
                    'index': input.index
                }));

            let wasItFoundInSelectedTransactions = R.not(R.isNil(findInputTransactionInTransactionList(inputTransactionsInTransa
            let wasItFoundInBlocks = R.not(R.isNil(findInputTransactionInTransactionList(inputTransactionsInTransaction(transact

            return wasItFoundInSelectedTransactions || wasItFoundInBlocks;
```

```
if (transactions.length > 0) {
    let feeTransaction = Transaction.fromJson({
        id: CryptoUtil.randomId(64),
        hash: null,
        type: 'fee',
        data: {
            inputs: [],
            outputs: [
                {
                    amount: Config.FEE_PER_TRANSACTION * transactions.length,
                    address: feeAddress,
                }
            ]
        }
    });

    transactions.push(feeTransaction);
```

4. Record of Student Attendance

Record student's attendance by student's `User_ID`, event's `eventID` and signature generated by user's secret key and store the record in the blockchain.

4.1 Implementation Logic: API Design

- **Endpoint**: `/operator/attendance`.
- **Request method**: `POST`
- **Input data:**

```
{
    "User_ID": "student123",
    "eventID": "event456",
    "privateKey": "private_key_string"
}
```

4.2 Implementation Logic: Main Code

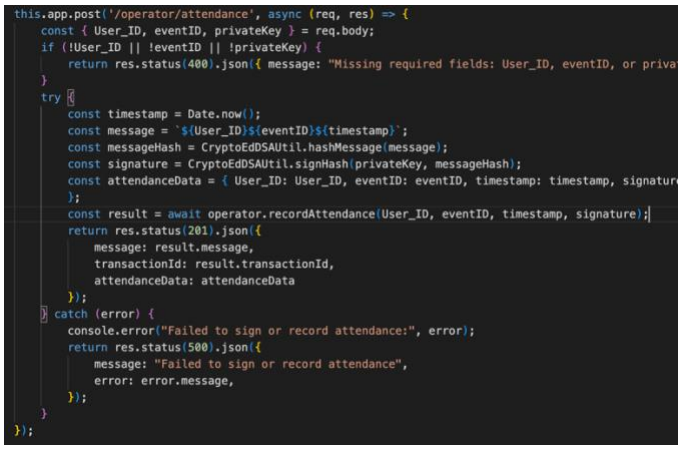
1. Transaction construction: The id generation uses ${User_ID}_${eventID}_${timestamp} to ensure that each record is unique.

2. The transaction is written to the blockchain using `this.blockchain.addTransaction(attendanceTransaction)` to add the transaction to the blockchain system. If there is an error while joining a transaction, an exception is caught, and an error log is printed, and a new error is thrown. When attendance is successfully recorded, a log is printed, and a success message and transaction ID are returned.

```
async recordAttendance(User_ID, eventID, timestamp, signature) {
    const attendanceTransaction = {
        id: `${User_ID}_${eventID}_${timestamp}`,
        type: 'attendance',
        data: {
            inputs: [],
            outputs: [
                {
                    User_ID: User_ID,
                    eventID: eventID,
                    timestamp: timestamp,
                    signature: signature,
                    amount: 0,
                    address: NaN
                }
            ]
        }
    };

    try {
        await this.blockchain.addTransaction(attendanceTransaction);
        console.log(`Attendance recorded for student ID: ${User_ID}, event ID: ${eventID} at timestamp: ${timestamp}`);

        return {
            message: "Attendance recorded successfully",
            transactionId: attendanceTransaction.id
        };
    } catch (error) {
        console.error("Failed to record attendance:", error.message);
        throw new Error("Failed to record attendance");
    }
}
```

Checks that the request body contains the required parameters: User_ID, eventID and

privateKey. If they are missing, a `400 status` code and an error message will be returned. Use CryptoEdDSAUtil tool library to generate signature. Call `operator.recordAttendance` , passing user ID, event ID, timestamp and signature. After the result is captured, it is returned encapsulated in the HTTP response. If something goes wrong at any stage (missing parameter, signature failure, logging failure, etc.), catch the exception and return a `500 status` code along with an error message.



```
this.app.post('/operator/attendance', async (req, res) => {
    const { User_ID, eventID, privateKey } = req.body;
    if (!User_ID || !eventID || !privateKey) {
        return res.status(400).json({ message: "Missing required fields: User_ID, eventID, or priva
    }
    try {
        const timestamp = Date.now();
        const message = `${User_ID}${eventID}${timestamp}`;
        const messageHash = CryptoEdDSAUtil.hashMessage(message);
        const signature = CryptoEdDSAUtil.signHash(privateKey, messageHash);
        const attendanceData = { User_ID: User_ID, eventID: eventID, timestamp: timestamp, signatur
        };
        const result = await operator.recordAttendance(User_ID, eventID, timestamp, signature);
        return res.status(201).json({
            message: result.message,
            transactionId: result.transactionId,
            attendanceData: attendanceData
        });
    } catch (error) {
        console.error("Failed to sign or record attendance:", error);
        return res.status(500).json({
            message: "Failed to sign or record attendance",
            error: error.message,
        });
    }
});
```

5. Searching attendance records by condition

Query student's attendance record by `User_ID`, `eventID` or time range.

1. API Design

Endpoint: `/blockchain/transactions/attendance`

Request method: `GET`

Input Parameters.

`User_ID`: (optional) Student ID.

`eventID`: (optional) Event ID.

`startTime` and `endTime`: (optional) Time range (any time format).

2. Filtering Logic

Iterates through all transactions in the blockchain and filters out those with `type` of `attendance`.

Match based on the conditions of the request parameters:

`User_ID/eventID` match: filter records in `outputs` where `User_ID/ eventID` is the same as the query value.

Time Range Matching: Filters records with `timestamp` in the `startTime` and `endTime` range.

```
getAttendanceTransactions({ User_ID, eventID, startTime, endTime }) {
    const transactionsInBlocks = R.flatten(R.map(R.prop('transactions'), this.getAllBlocks()));
    const parsedStartTime = startTime ? new Date(startTime).getTime() : null;
    const parsedEndTime = endTime ? new Date(endTime).getTime() : null;

    return transactionsInBlocks.filter(tx => {
        if (tx.type !== 'attendance') return false;

        const isStudentMatch = !User_ID || R.propEq('User_ID', User_ID, tx.data);
        const isEventMatch = !eventID || R.propEq('eventID', eventID, tx.data);
        const isTimeMatch = (!parsedStartTime || tx.data.timestamp >= parsedStartTime) &&
            (!parsedEndTime || tx.data.timestamp <= parsedEndTime);

        return isStudentMatch && isEventMatch && isTimeMatch;
    });
}
```

Store results: Returns `attendance` transactions that match the conditions, for example:

```
{
    "type": "attendance",
    "data": {
        "inputs": [],
        "outputs": [
            {
                "User_ID": "student123",
                "eventID": "event456",
                "timestamp": 1732780800000,
                "signature": "generated_signature_string"
            }
        ]
    }
}
```

6. List All Existing Event

```
getAllEvents() {
    console.log('getAllEvents called');
    const transactionsInBlocks = R.flatten(R.map(R.prop('transactions'), this.getAllBlocks()));
    const events = transactionsInBlocks
        .filter(tx => tx.type === 'event')
        .map(tx => tx.data.outputs[0])
        .map(output => ({
            eventID: output.eventID
        }));
    console.log('All events:', events);
    return events;
}
```

Calls this.getAllBlocks() to get all blocks in the blockchain. Extracts the 'transactions' field from each block using R.map(R.prop('transactions'), ...) and flattens the resulting

nested array using R.flatten(...). Filters the transactions to include only those where tx.type = 'event'. Maps the filtered transactions to retrieve the first output in tx.data.outputs and extracts its eventID.

```
this.app.get('/blockchain/transactions/events', (res) => {
    try {
        const events = blockchain.getAllEvents();
        if (events.length === 0) {
            return res.status(404).json({ message: 'No events found' });
        }
        res.status(200).json(events);
    } catch (error) {
        console.error('Error fetching events:', error.message);
        res.status(500).json({ message: 'Failed to fetch events', error: error.message });
    }
});
```

Calls blockchain.getAllEvents() to retrieve the events. Checks if the events array is empty. If events.length = 0, it returns a 404 status with the message 'No events found'. Otherwise, responds with a 200 status and the list of events.

7. Retrieve Secret Key by Wallet ID and Password (Improvements Discussed Later)

```
this.app.post('/operator/getSecretKey', (req, res) => {
    const { walletId, password } = req.body;

    if (!walletId || !password) {
        return res.status(400).json({ message: "Missing walletId or password" });
    }
    try {
        const passwordHash = CryptoUtil.hash(password); // Hash the password
        const wallet = operator.getWalletById(walletId);
        if (!wallet) {
            return res.status(404).json({ message: "Wallet not found" });
        }
        if (wallet.passwordHash !== passwordHash) {
            return res.status(403).json({ message: "Invalid password" });
        }
        // Retrieve the first secretKey from the keyPairs array
        if (!wallet.keyPairs || wallet.keyPairs.length === 0) {
            return res.status(404).json({ message: "No key pairs found in the wallet" });
        }
        const secretKey = wallet.keyPairs[0].secretKey;
        res.status(200).json({ secretKey });
    } catch (error) {
        console.error("Error retrieving secret key:", error.message);
    }
});
```

Input Validation: Checks if the walletId and password are provided in the request body. If any parameter is missing, returns a 400 status code with the message "Missing walletId or password".

Password Hashing: Uses CryptoUtil.hash to hash the provided password.

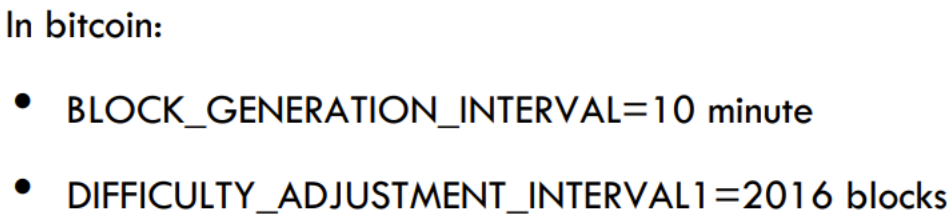
Retrieve Wallet: Calls operator.getWalletById(walletId) to fetch the wallet data associated with the given walletId. If the wallet is not found, returns a 404 status code with the message "Wallet not found".

Password Verification: Compares the hashed password (passwordHash) with the stored hash (wallet.passwordHash). If the passwords do not match, returns a 403 status code with the message "Invalid password".

Retrieve Secret Key: Checks if the keyPairs array exists and has at least one key pair.If no key pairs are found, returns a 404 status code with the message "No key pairs found in the wallet". Otherwise, retrieves the first secretKey from the keyPairs array and includes it in the response.

8. Dynamic Difficulty

We implement a mechanism to adjust the difficulty level of block mining dynamically. This is based on the time taken to generate the previous set of blocks (we set 10 this time, but any number can be chosen) similar to the approach used in Bitcoin or Ethereum.



First, we introduce a new property called 'difficulty' for each block, making it easier to determine the difficulty based on time:

{
"index": 8,
…..
"timestamp": 1732868603.341,
……

"difficulty": 9007199254740991

}

We adjusted ‘addBlock’ function in ‘index.js’ of ‘blockchain’ to achieve it:

```
async addBlock(newBlock, emit = true) {
    newBlock.difficulty=this.getDifficulty(newBlock.index);
```

In ‘config.js’, we set the BLOCK_GENERATION_INTERVAL to 10seconds, which means we want to generate a block every 10 seconds; moreover, we set DIFFICULTY_ADJUSTMENT_INTERVAL to 10, which means we adjust the difficulty every 10 blocks. These parameters can be set by hands:

```
const EVERY_X_BLOCKS = 10; //DIFFICULTY_ADJUSTMENT_INTERVAL
const TARGET_TIME = 10; //BLOCK_GENERATION_INTERVAL
```

We aim to avoid making mundane changes to the difficulty level. Therefore, we will assess whether an update is necessary only once every 10 blocks. If no adjustment is needed, the new block will retain the same difficulty as the previous one.

```
if (blocks.length <= EVERY_X_BLOCKS){
    return BASE_DIFFICULTY;
}

if ((index + 1) % EVERY_X_BLOCKS !== 0) {
    return blocks[blocks.length-1].difficulty;
}
```

When we adjust the difficulty, we increase it if the time taken is less than the expected duration and decrease it if the time taken is greater than expected. We calculate the averageTime based on the last 10 blocks. To adjust the current difficulty, we multiply the current difficulty by (TARGET_TIME / averageTime). If averageTime is less than TARGET_TIME, the result will be greater than 1, indicating that we should increase the difficulty. Conversely, if averageTime is greater than

TARGET_TIME, we will decrease the difficulty.

```
const adjustmentStartIndex = Math.max(0, blocks.length - EVERY_X_BLOCKS);
const timeSpent = blocks[blocks.length-1].timestamp - blocks[adjustmentStartIndex].timestamp;
const averageTime = timeSpent / EVERY_X_BLOCKS;
const currentDifficulty = blocks[blocks.length-1].difficulty;
  console.log(`average of time ${averageTime} target time ${TARGET_TIME}, currentDifficulty ${

let newDifficulty;
newDifficulty = currentDifficulty * (TARGET_TIME / averageTime);

return Math.max(newDifficulty, 1);
```

The difficulty changes every 10 blocks:

9. Fork Resolution

To resolve a fork in a blockchain, common measures were changing the chain according to length or difficulty. Length based replace chain function already exists; therefore, we kept that and mainly focused on the development of difficulty-based fork resolution.

From dynamic difficulty in the previous part, we could thereby get the cumulative difficulty, like below.

```
calculateCumulativeDifficulty(blocks) {
    return R.sum(R.map(block => Math.pow(2, block.difficulty), blocks));
}
```

From cumulative difficulty, we can construct an architecture of fork resolution. When adding any blocks, we resolve if there were any fork exist by detectForks().

```
detectForks() {
    const forkedBlocks = {};

    // Group blocks by their `previousHash`
    this.blocks.forEach((block) => {
        if (!forkedBlocks[block.previousHash]) {
            forkedBlocks[block.previousHash] = [];
        }
        forkedBlocks[block.previousHash].push(block);
    });

    // Detect forks: any `previousHash` with more than one child block
    const forks = Object.keys(forkedBlocks).filter(hash => forkedBlocks[hash].length > 1);

    if (forks.length === 0) {
        console.info('No forks detected.');
        return [];
    } else {
        console.warn(`Forks detected at blocks: ${forks.join(', ')}`);
    }

    // Get the full chains for each fork
    const forkChains = forks.map(hash => forkedBlocks[hash].map(block => this.getChainFromBlock(block)));

    return forkChains.flat(); // Flatten the array if there are multiple forks
}
```

This function check fork(s) by checking if any `previousHash`(block) with more than one child block, if fork exists, call getChainfromBlock() to obtain multiple chains prepared for comparison.

```
// Method to get the chain from a specific block
getChainFromBlock(block) {
    const chain = [];
    let currentBlock = block;

    while (currentBlock) {
        chain.push(currentBlock);
        currentBlock = this.blocks.find(b => b.hash === currentBlock.previousHash);
    }

    return chain.reverse(); // Reverse to return the chain from genesis to the given block
}
```

The function constructs a blockchain-like sequence starting from a given block and traces back to the genesis block.

With forks detected and chains were prepared, in resolveFork(), we compared two chains with its cumulative difficulty, mentioned previously calculateCumulativeDifficulty(). We choose the chain with the largest cumulative difficulty to be the bestChain, and relocate the pointer this.blocks pointing to the last block of bestChain.

```
resolveFork() {
    const forks = this.detectForks();
    if (forks.length === 0) {
        console.info('No forks to resolve.');
        return;
    }

    // Select the best chain based on cumulative difficulty
    const bestChain = forks.reduce((best, currentChain) => {
        const bestDifficulty = this.calculateCumulativeDifficulty(best);
        const currentDifficulty = this.calculateCumulativeDifficulty(currentChain);
        return currentDifficulty > bestDifficulty ? currentChain : best;
    });


    console.info(`Best chain selected with cumulative difficulty: ${this.calculateCumulativeDifficulty(bestChain)}`);

    // Replace the chain
    try {
        this.blocks = bestChain; // Replace the current chain with the new chain
        console.info('Fork resolved. Chain replaced successfully.');
        // Update transaction pool to remove confirmed transactions
        this.updateTransactionPool();
    } catch (error) {
        console.error('Failed to resolve fork:', error);
    }
}
```

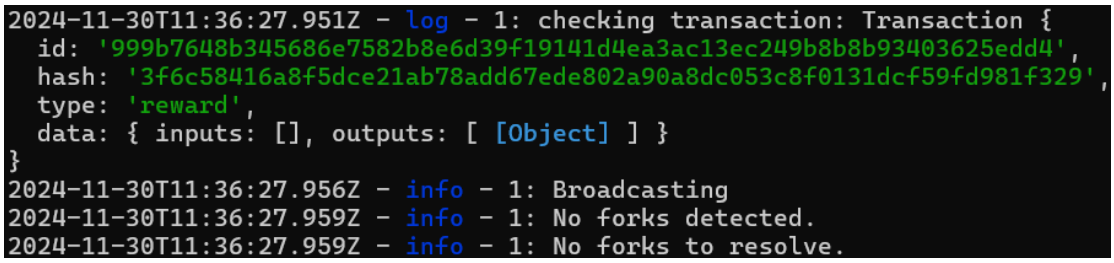
With the chain replaced, we finally called updateTransactionPool() to update transactions shown.

```
updateTransactionPool() {
    const allTransactionsInChain = R.flatten(
        this.blocks.map(block => block.transactions)
    );

    // Filter out confirmed transactions from the transaction pool
    this.transactions = this.transactions.filter(
        transaction => !allTransactionsInChain.find(
            chainTx => chainTx.id === transaction.id
        )
    );

    this.transactionsDb.write(this.transactions); // Persist changes to the database
    console.info('Transaction pool updated.');
}
```

And in the console page, we can check the fork working progress.

```
2024-11-30T10:59:03.582Z - log - 1: checking transaction: Transaction {
  id: '4ad2dac81573336def02492c887d5b7e701a5e9cef07e79932b5fab8c89c43dd',
  hash: '78817c801593c9cbaefa21c9598f91d00b3c3be7aa6eea812e0bcb2742721b91',
  type: 'reward',
  data: { inputs: [], outputs: [ [Object] ] }
}
2024-11-30T10:59:03.587Z - info - 1: Broadcasting
2024-11-30T10:59:03.588Z - warn - 1: Forks detected at blocks: 105cf137794c814fb168dbfaa29b82dd5adbebdc44bbb7b72a5eaa060
2c86de2
2024-11-30T10:59:03.589Z - info - 1: Best chain selected with cumulative difficulty: Infinity
2024-11-30T10:59:03.589Z - info - 1: Fork resolved. Chain replaced successfully.
2024-11-30T10:59:03.591Z - info - 1: Transaction pool updated.
2024-11-30T10:59:03.591Z - info - 1: Transaction Pool updated.
```

(for test we set the difficulty extremely high, so in here it will return cumulative difficulty as Infinity)



(no forks were detected)

10. Front-end Interactive Interface

The front-end of the project is developed using **React**, a popular JavaScript library for building user interfaces. We chose React because of its component-based architecture, virtual DOM for efficient updates, and strong ecosystem support. React allows us to build reusable components that enhance development speed and maintainability. Additionally, **React Router** is integrated to facilitate navigation and ensure a modular code structure.

By using React, we achieved a clean, efficient, and user-friendly front-end that is seamlessly integrated with the backend. This setup not only simplifies development but also ensures a robust user experience.

10.1 Why React Framework?

1. **Component-Based Architecture**: React allows developers to divide the UI into reusable components, reducing redundancy and improving readability. Components such as `Layout.js` and `Sidebar.js` can be reused across multiple pages, ensuring consistency.
2. **Virtual DOM**: React's virtual DOM ensures efficient UI rendering by updating only the components that change, leading to faster performance compared to traditional frameworks.
3. **React Router**: React Router simplifies navigation within the app. It allows users to move between different views (pages) seamlessly, such as `attendance`, `event`, and `query`.
4. **Ecosystem and Community**: React has a vast library ecosystem, including tools like `react-datepicker` for date inputs and `axios` for API interactions, which accelerated development.
5. **Ease of State Management**: React's state hooks simplify managing dynamic data like user inputs or API responses.

6. **Scalability**: React's modular approach makes it easier to scale the application with new features and functionalities.

10.2 How we Setting up the React Project in the begining

1. **Initialize the Project**: This command sets up the project structure and installs essential dependencies for React.

   npx create-react-app client

2. **Install Required Dependencies**:

   ```
   npm install axios react-router-dom react-datepicker
   ```

   These dependencies enable HTTP requests, routing, and date selection functionality, essential for building interactive features.

3. **Enable Cross-Origin Requests**: To allow the front-end running on `localhost:3000` to communicate with the backend server, **CORS (Cross-Origin Resource Sharing)** is enabled in the backend using the following code:

   ```
   const cors = require('cors');
   ```

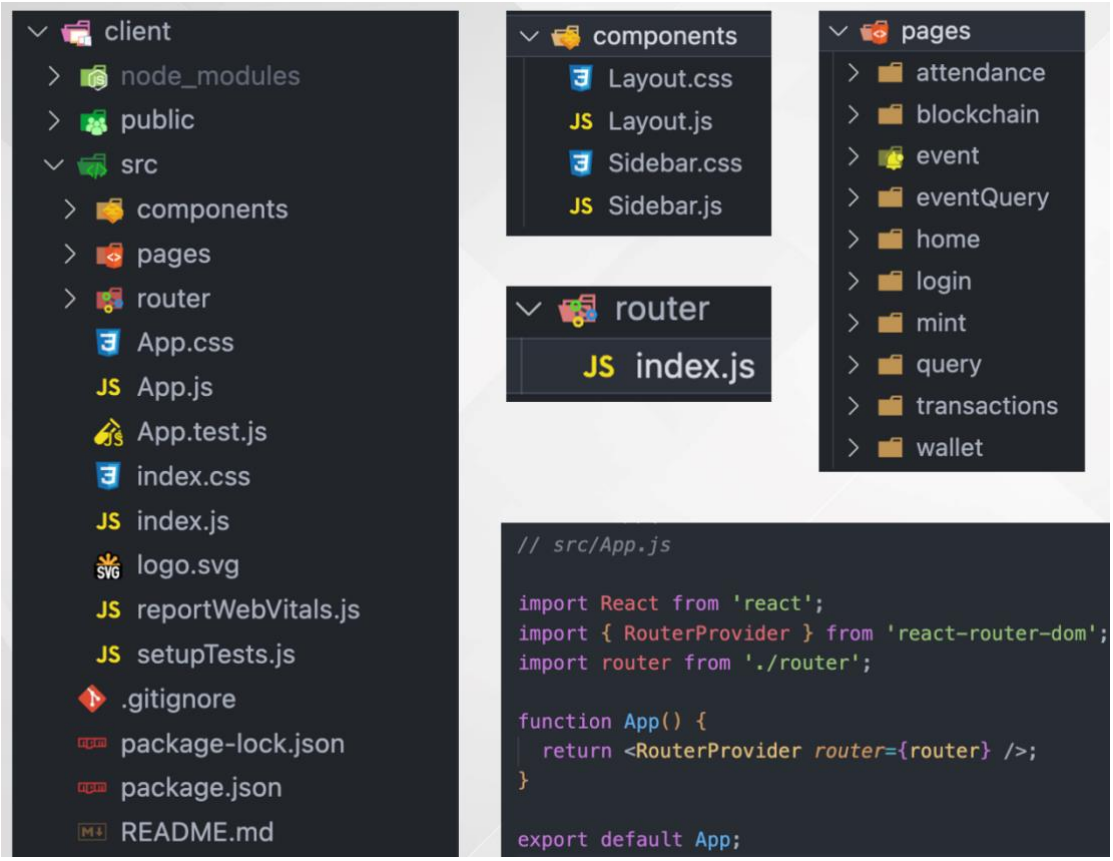
   ```
   this.app = express();
   this.app.use(cors({
       origin: 'http://localhost:3000',
       methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
       credentials: true,
       allowedHeaders: 'Content-Type, Authorization'
   }));
   ```

4. **Start the Development Server**: Once the setup is complete, the React development server is started with:

   ```
   cd client

   npm start
   ```

   The app runs at `http://localhost:3000`, where you can test and debug the front-end.

10.3 Front-End Structure



The project is organized into the following directories:

1. **components**: Contains reusable UI components like `Layout.js` and `Sidebar.js`. Ensures modular design and consistency.
2. **pages**: Includes individual views such as `attendance`, `event`, `query`, etc. Each page is designed to handle specific application features.
3. **router**: Manages routing configurations using React Router. The `router/index.js` file contains all route definitions, enabling easy navigation between pages.
4. **App.js**: The `App.js` file integrates React Router for handling navigation across pages. Acts as the entry point for the React application. Using `RouterProvider`, it connects the app to the routes defined in `router/index.js`. The code snippet below demonstrates this setup:

10. 4 Explanation of Code: `RegistrationForm.js`

Since the functions and implementations of each part have a lot in common, I will use RegistrationForm.js to describe the main functions I implemented and how to connect and interact with the backend.

The `RegistrationForm.js` component handles the **user registration process** by interacting with the backend to register a new student and retrieve their wallet details. The implementation uses **React**, **Ant Design**, and **Axios** for creating the user interface, managing states, and making API calls respectively.

10.4.1 Key Functionalities

1. **State Management**:
    - `registrationResult`: Stores the successful registration response (User ID, Wallet ID, Public Key).
    - `error`: Captures error messages if the API call fails.
    - `isSubmitting`: Indicates whether the form is currently submitting to disable the button and show a loading state.
2. **Form Submission**: When the user submits the form, the `onFinish` function is triggered. It sends a POST request with the input values (`User_ID` and `password`) to the backend API endpoint (`/operator/registerStudent`).
3. **Backend Communication**: **Axios** is used to make HTTP POST requests.
    - If the request succeeds, the `registrationResult` state is updated with the response data.
    - If the request fails, the `error` state is updated to display an error message.
4. **Form Design**: Built using **Ant Design's** `Form`, `Input`, `Button`, and `Alert` components to provide a modern and user-friendly design. Validation rules ensure mandatory fields are filled before submission.
5. **UI Feedback**: Displays a success message with the user's details (User ID, Wallet ID, Public Key) if registration is successful. Shows an error message if registration fails.

10.4.2 Corresponding Backend Integration (Connection and Interaction)

1. **Frontend API Call**: The frontend sends the POST request to the backend using the following code:

```
const onFinish = async (values) => {
    setIsSubmitting(true);
    setError(null);
    setRegistrationResult(null);

    try {
        const response = await axios.post('http://localhost:3001/operator/registerStudent', values);
        setRegistrationResult(response.data);
    } catch (err) {
        console.error(err);
        setError(err.response?.data?.message || 'Registration failed, please try again later.');
    } finally {
        setIsSubmitting(false);
    }
};
```

2. **Backend Endpoint**: The backend Node.js server listens for the request on the `/operator/registerStudent` endpoint:

```
this.app.post('/operator/registerStudent', async (req, res) => {
    console.log("Register Student Route Accessed");
    const { User_ID, password } = req.body;

    try {
        const result = await operator.User_register(User_ID, password);
        res.status(201).json(result);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
});
```

```
lib > operator > JS index.js > Operator > User_register
13   class Operator {
91
92       async User_register(User_ID, password) {
93           const newWallet = this.createWalletFromPassword(password);
94           const publicKeyAddress = this.generateAddressForWallet(newWallet.id);
95
96           //console.log(`Student ${User_ID} registered as a miner with wallet ID: ${newWallet.id} and address: ${publicKeyAddress}`);
97           const registrationTransaction = {
98               id: `reg_${User_ID}`,
99               type: 'registration',
100              data: {
101                  inputs: [],
102                  outputs: [
103                      {
104                          User_ID: User_ID,
105                          publicKey: publicKeyAddress,
106                          amount: 0,
107                          address: NaN
108                      }
109                  ]
110              }
111          };
112          //console.log("test", registrationTransaction);
113
114          await this.blockchain.addTransaction(registrationTransaction);
115          console.log("newWallet:", newWallet);
116          console.log("publicKeyAddress:", publicKeyAddress);
117          console.log(`Student ${User_ID} registered with wallet ID: ${newWallet.id} and public address: ${publicKeyAddress}`);
118
119          return {
120              User_ID: User_ID,
121              walletID: newWallet.id,
122              publicKey: publicKeyAddress
123          };
124      }
```

3. **Backend Logic**: The `User_register` function generates a wallet for the student, returning the `User_ID`, `walletID`, and `publicKey` to the frontend. If any error occurs (e.g., missing fields or server issues), an appropriate error message is sent back to the frontend.
4. **API Response**: On successful registration, the backend responds with:

```
{
  "User_ID": "student123",
  "walletID": "wallet456",
  "publicKey": "publicKey789"
}
```

    - This response is displayed on the frontend under the "Registration Successful" section.

11. Challenge and Further Improvement:

After finishing presentation, we noticed that directly showing the plaintext of private key to user is dangerous. It is easy for attacker to get the information to forge a fake attendance recording. Therefore, we can further improve our project by either return the encrypted result of private key or let the whole process happened in server side. Here we will explain the second method to elaborate how to improve it. We use secret key to sign the attendance in server side instead of letting user type their private key. The detailed steps are as below:

1. Secure Wallet Creation (Wallet Creation - Secure Storage of Private Keys)

When the user creates a wallet, the server generates an **RSA** key pair (Public Key and Private Key).

The server uses the user's password to generate an encryption key via **Argon2** or **PBKDF2** (Argon2 is recommended because it is more resistant to GPU attacks).

This encryption key is used to encrypt the Private Key with **AES-256**, and the encrypted Private Key is stored in the database.

2. Initiating Attendance Transactions

User submits their **User ID**, **Password** and **Event ID** via the client API. The server verifies the user's identity: it uses the **Argon2** derived encryption key and decrypts the Private Key stored in the database. If the authentication passes, the server continues to the next step; otherwise, it returns an error.

3. Server-Side Signing

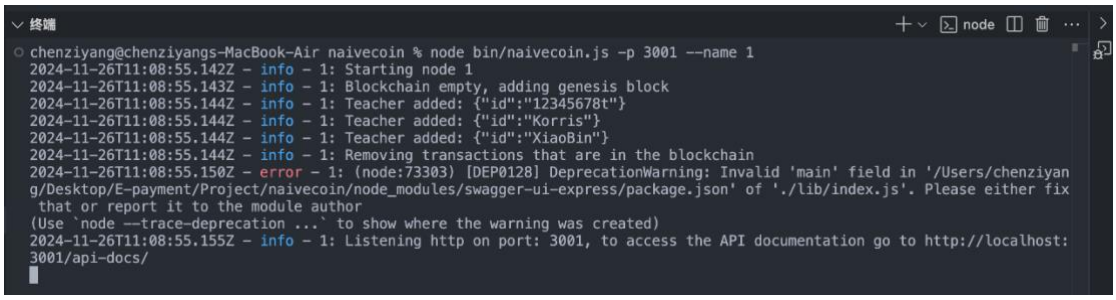
The server uses the decrypted **Private Key** to digitally sign the transaction using the **RSA** algorithm. The generated signature ensures that the transaction is authentic and tamper-proof. The server does not return the decrypted Private Key, but only the signed transaction record to the user.

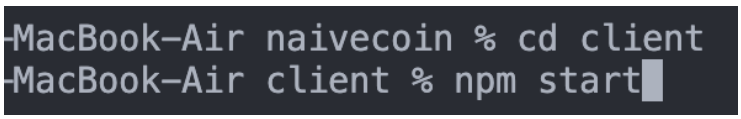
4. Preventing Duplicate Requests

The server adds a **UUID** or **random number** to each transaction record as a unique identifier. Before recording a transaction, the server checks the database or blockchain to ensure that there are no duplicate identifiers. The transaction ID is generated using the **SHA-256** hash function to ensure uniqueness and tamper resistance.

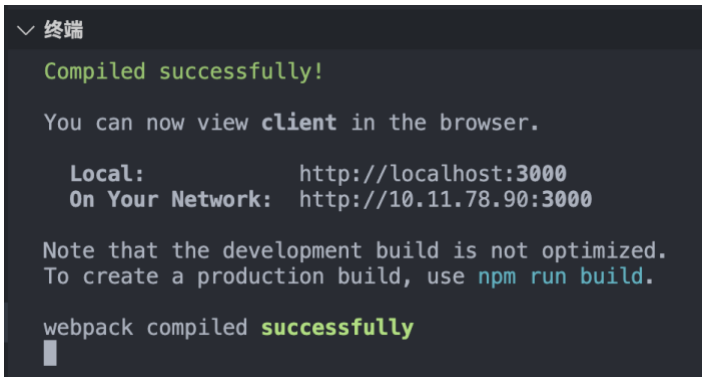
12. Test Cases

Initialize: input "node bin/naivecoin.js -p 3001 --name 1" in terminal (MacOS)



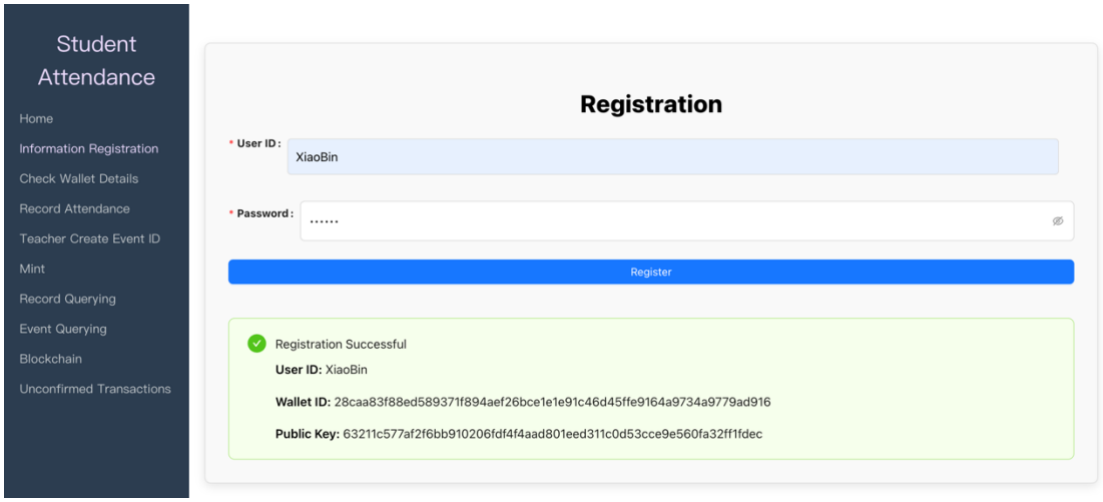


The program automatically starts.

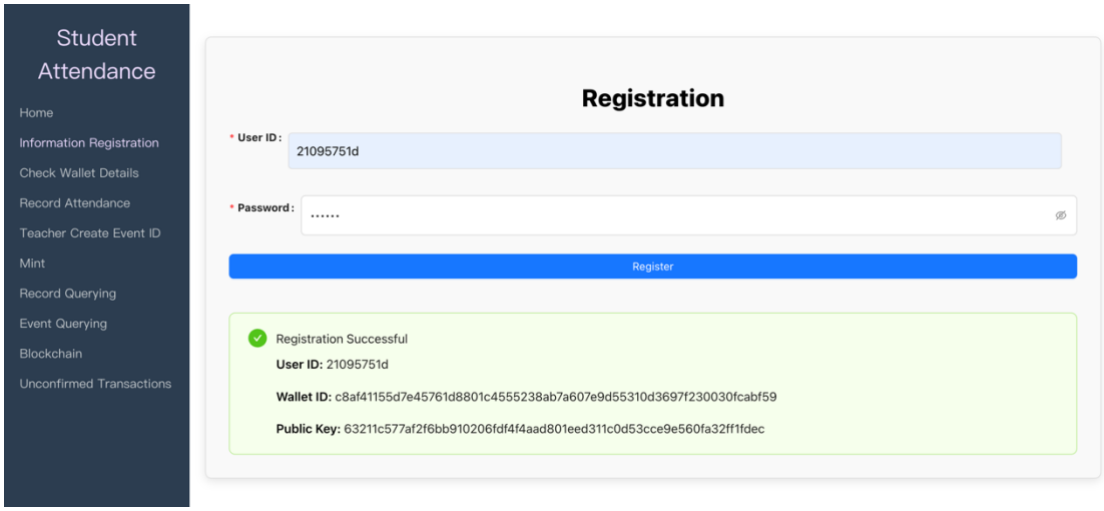


The Genesis block automatically generated:

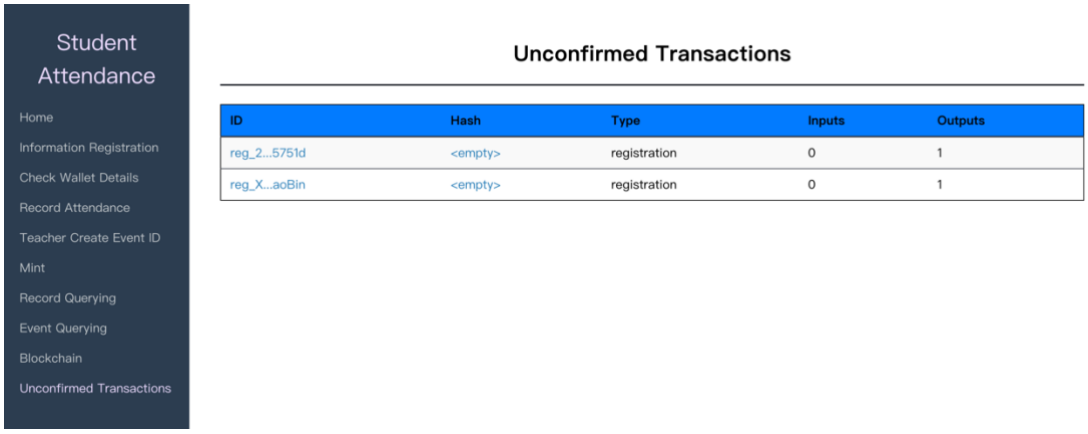
Teacher's Registration. The User's information is illustrated below:



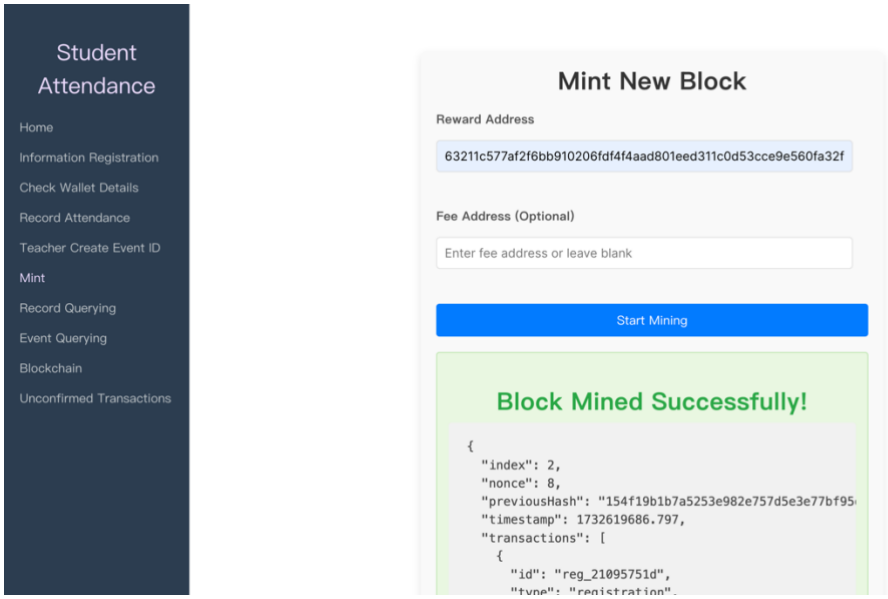
Student Registration:



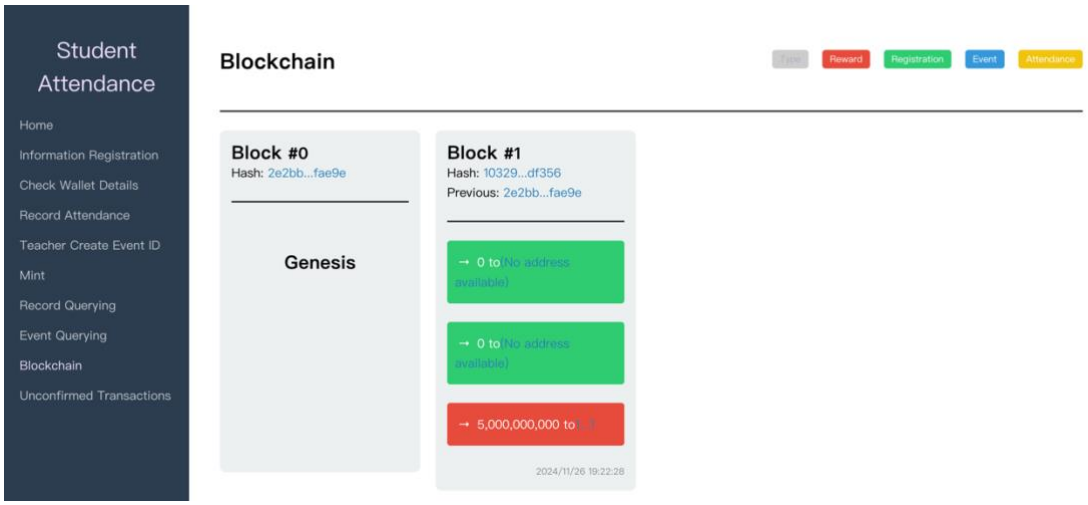
The registrations are listed as "Unconfirmed Transactions" before mining.



Every step should be mint (User registration, Creating event, Recording attendance)

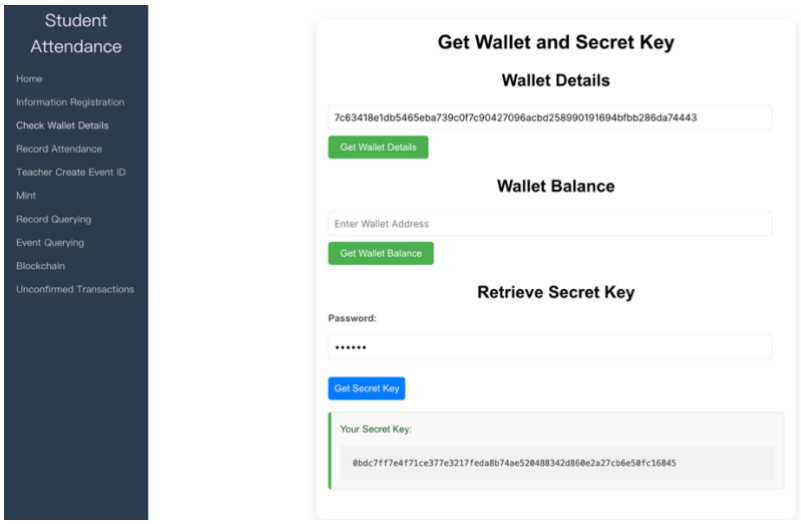


After mining, the information will be removed from “Unconfirmed Transactions” and stored in the blockchain.

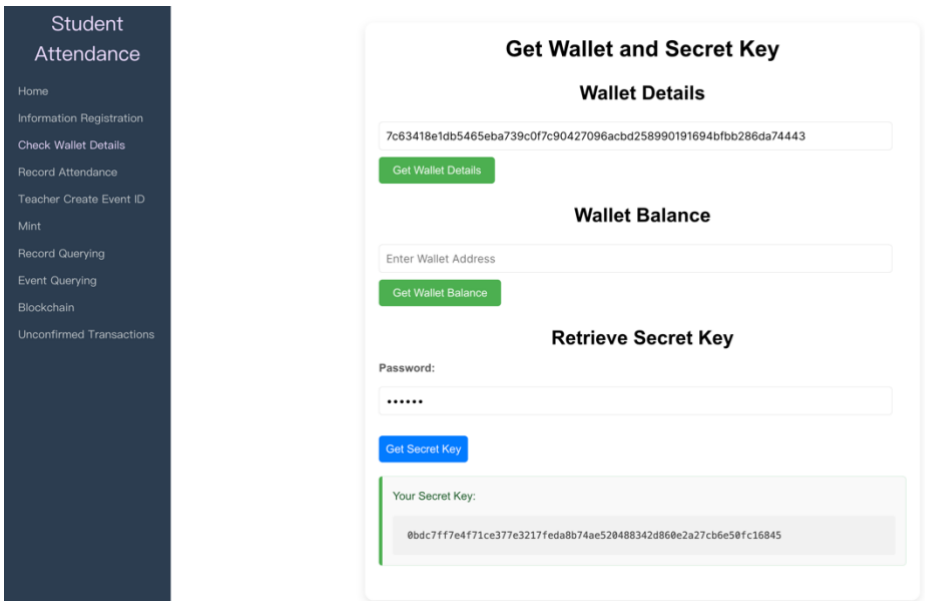


Secret key acquisition: Check Wallet Details, use the Wallet ID and password.

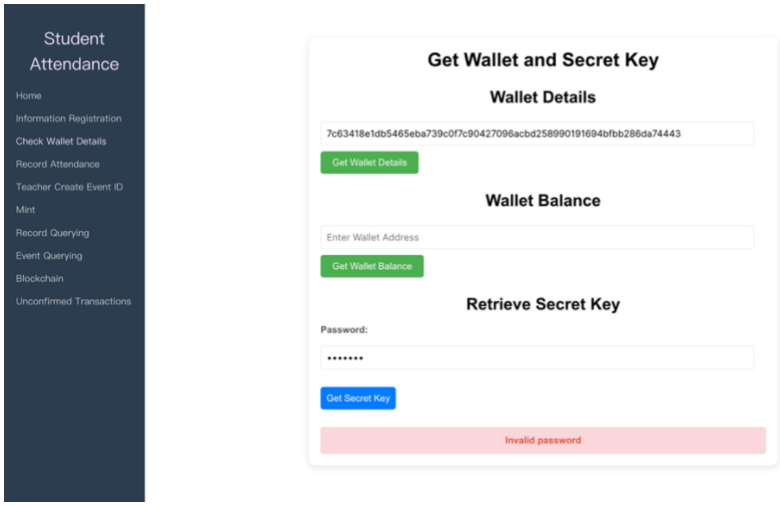
Secret Key for XiaoBin(teacher) is illustrated:



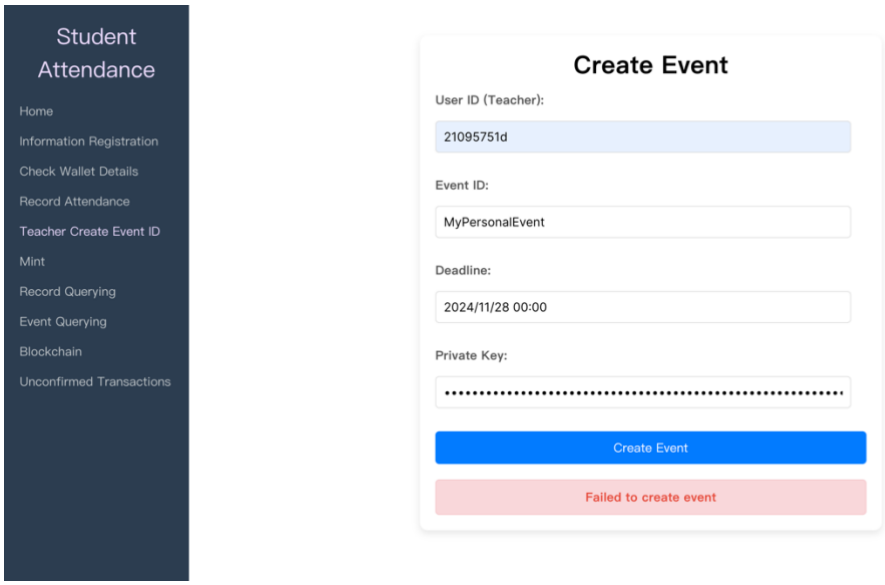
Secret Key for the student:



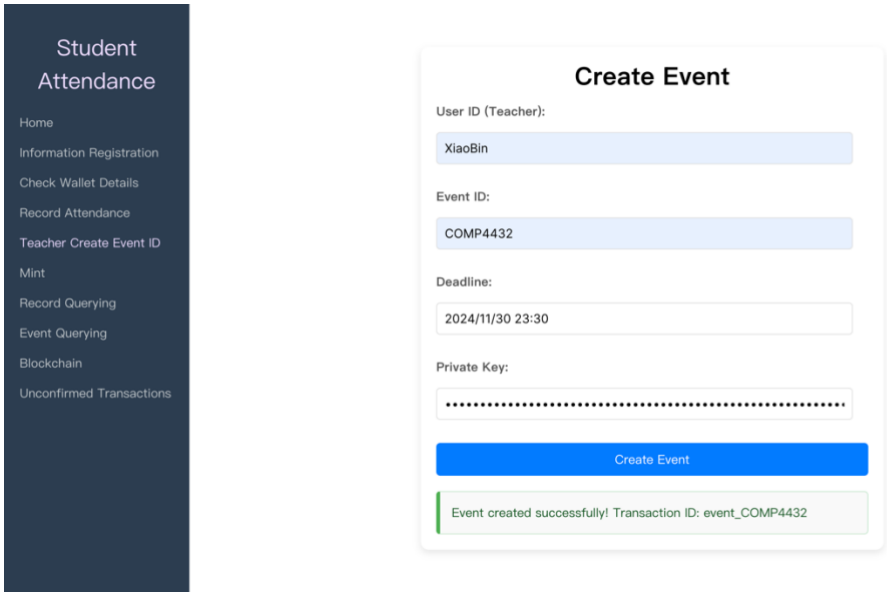
If user type wrong wallet ID or password, they will not get the private key:



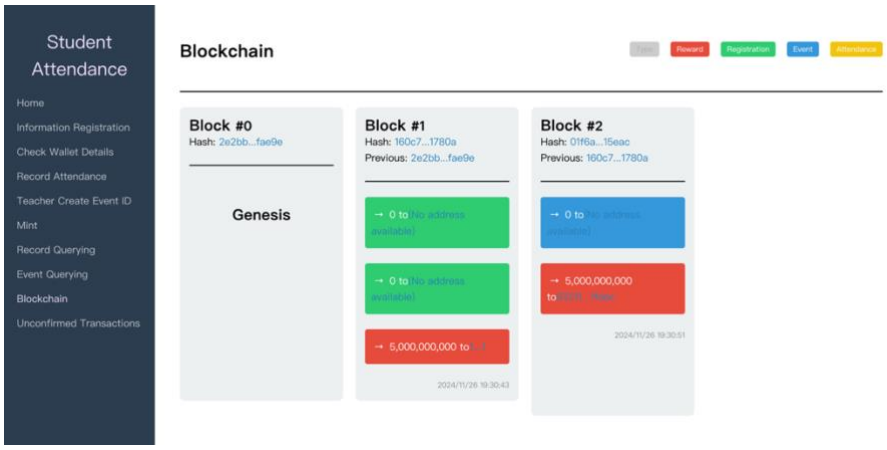
If an illegal user (e.g. student) want to create a new event:



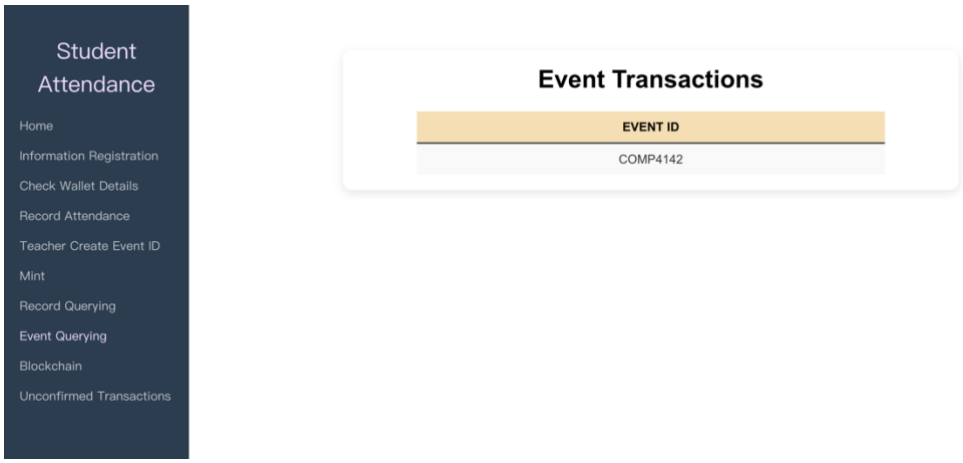
Event Creating:



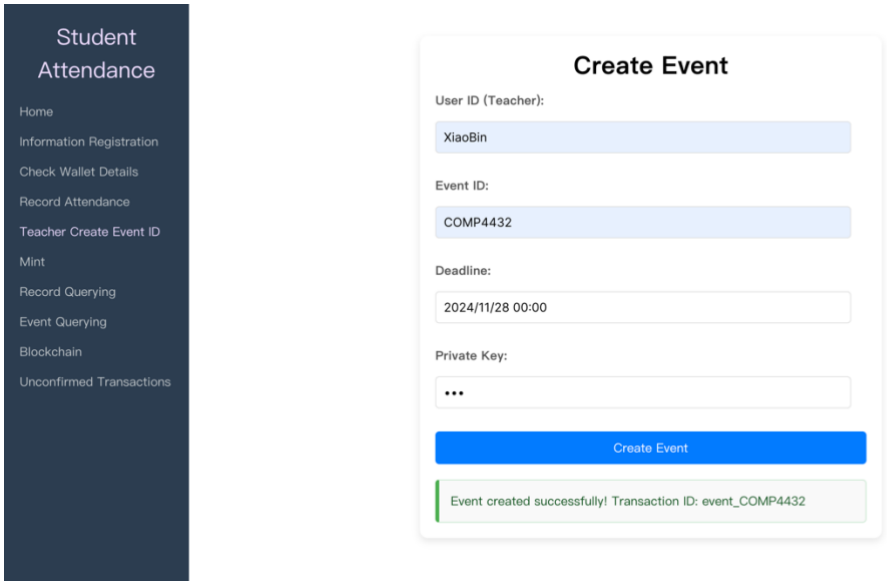
Stored in Blockchain



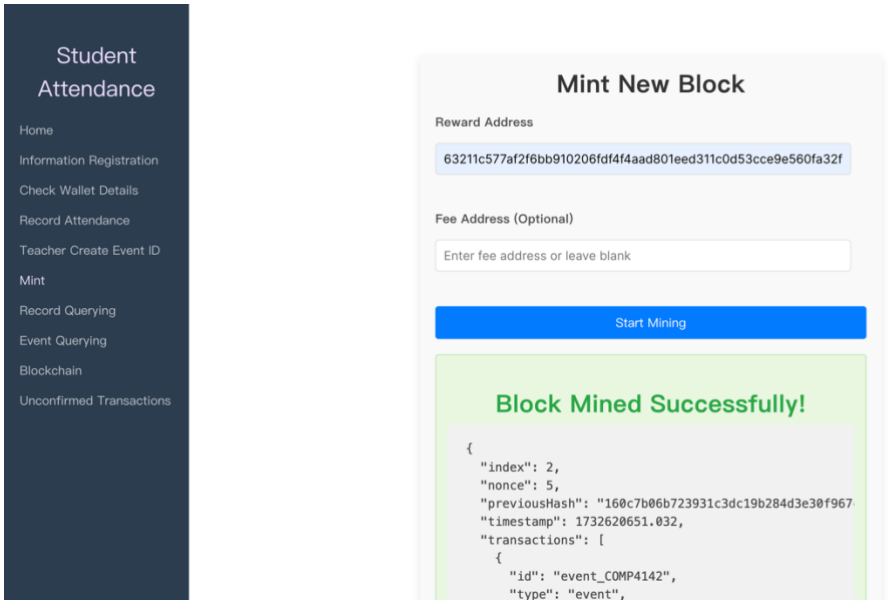
List all Event ID:

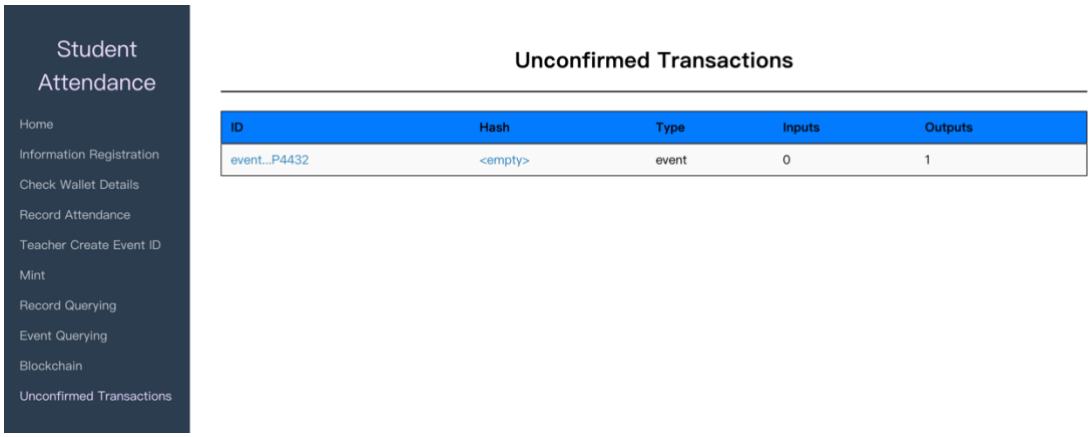


If we use a wrong private key to publish an event, the system still records the transaction:

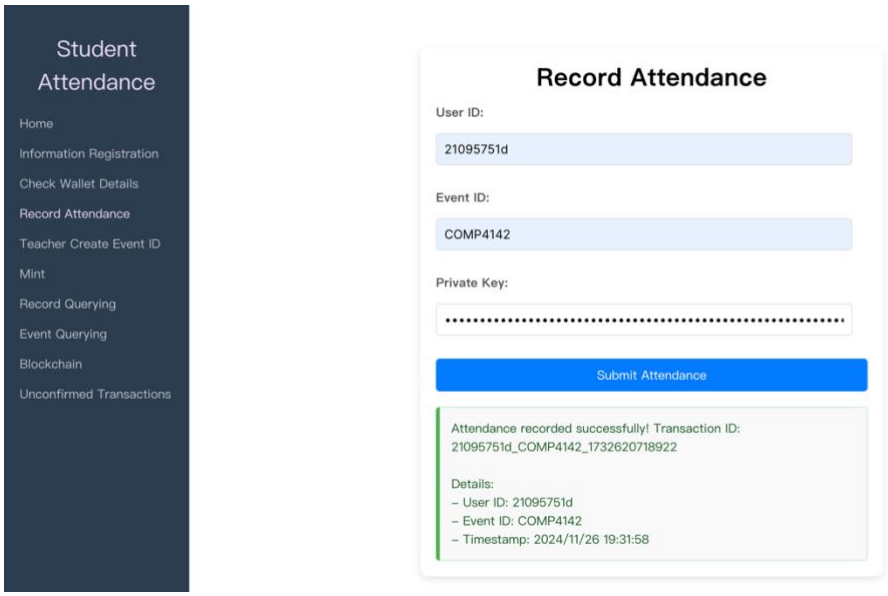


However, the event won't be mint and cannot be added into the blockchain:

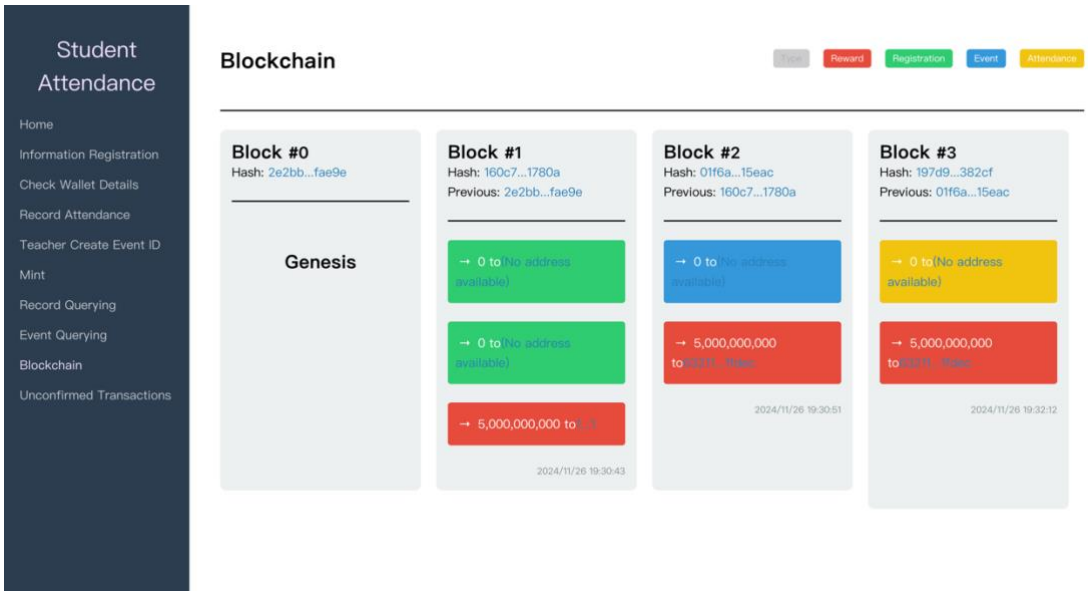




Attendance record for students:

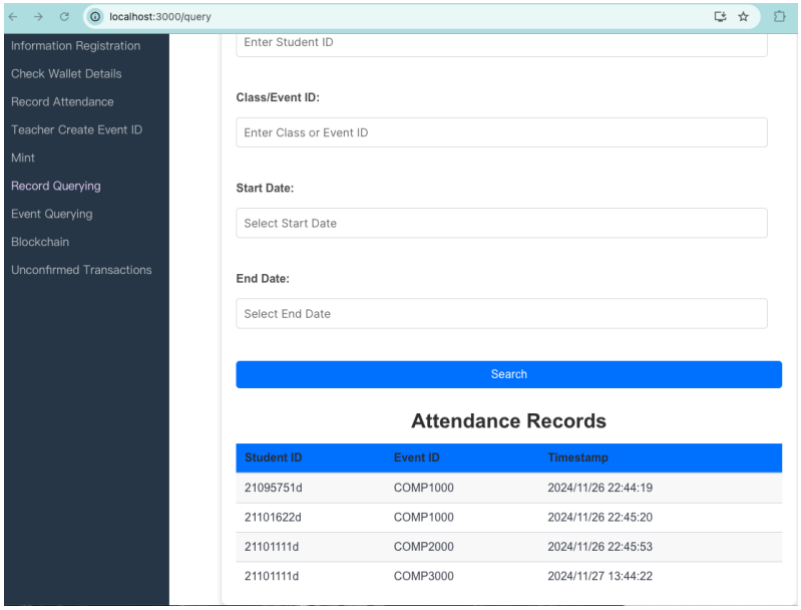


After mining the transaction:

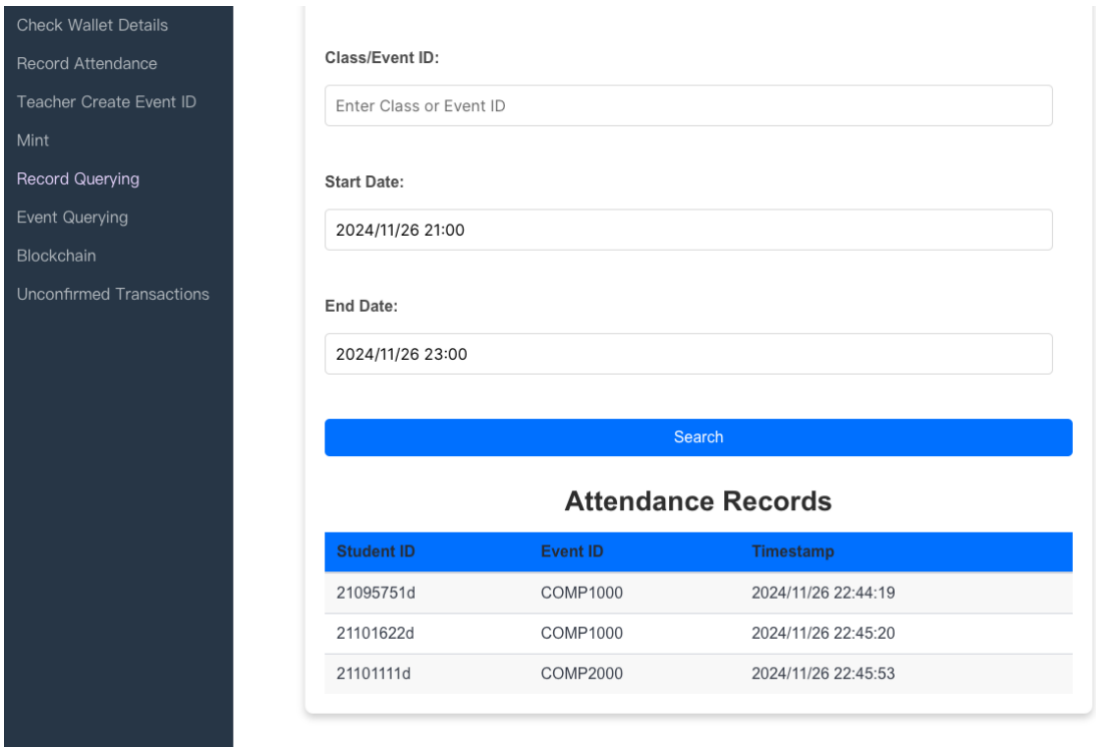


For query attendance recordings:

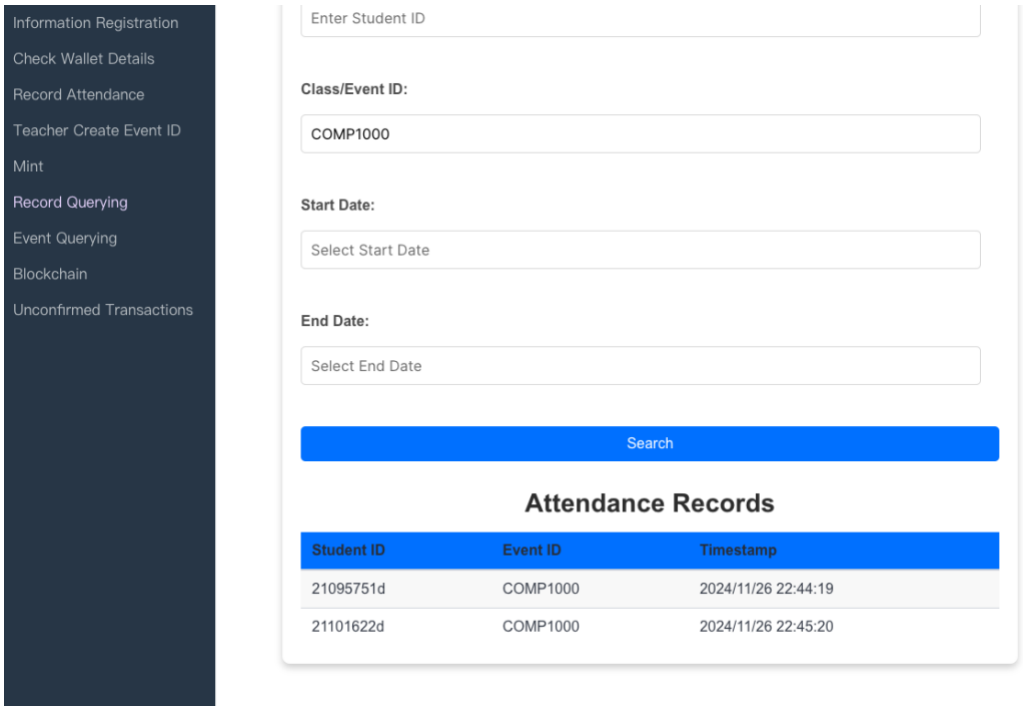
If we do not put anything and click “search”, the system will list all existing records:



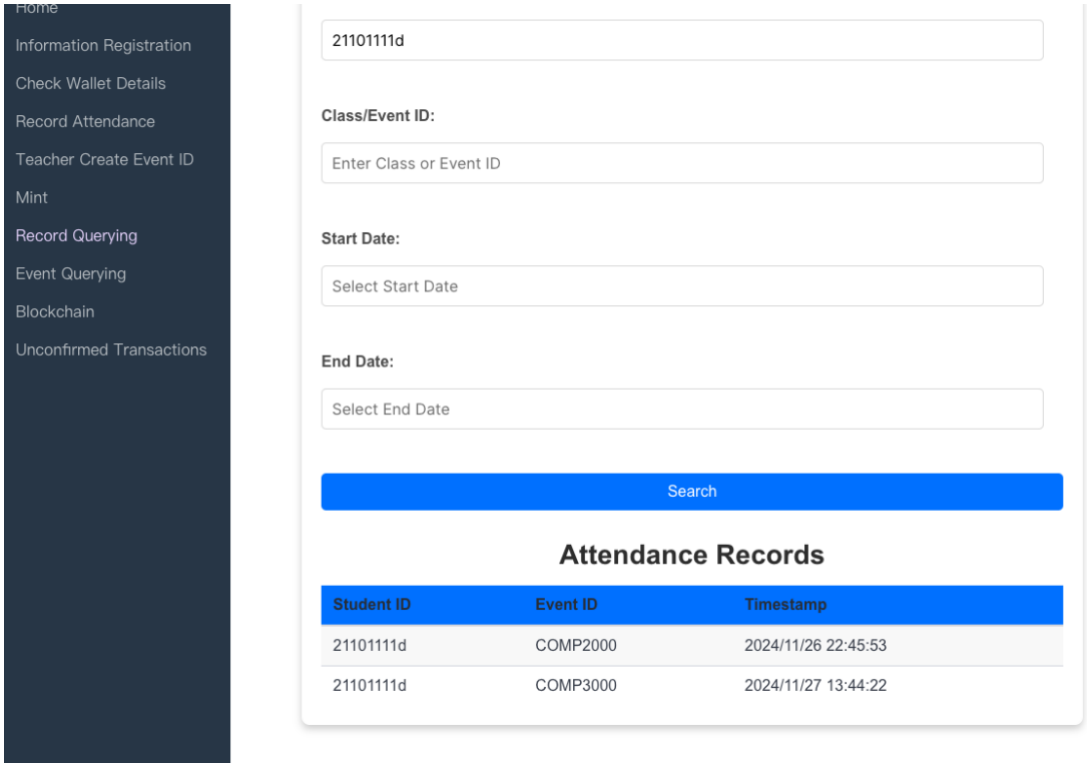
If we set a specific time period, we can find the corresponding records:



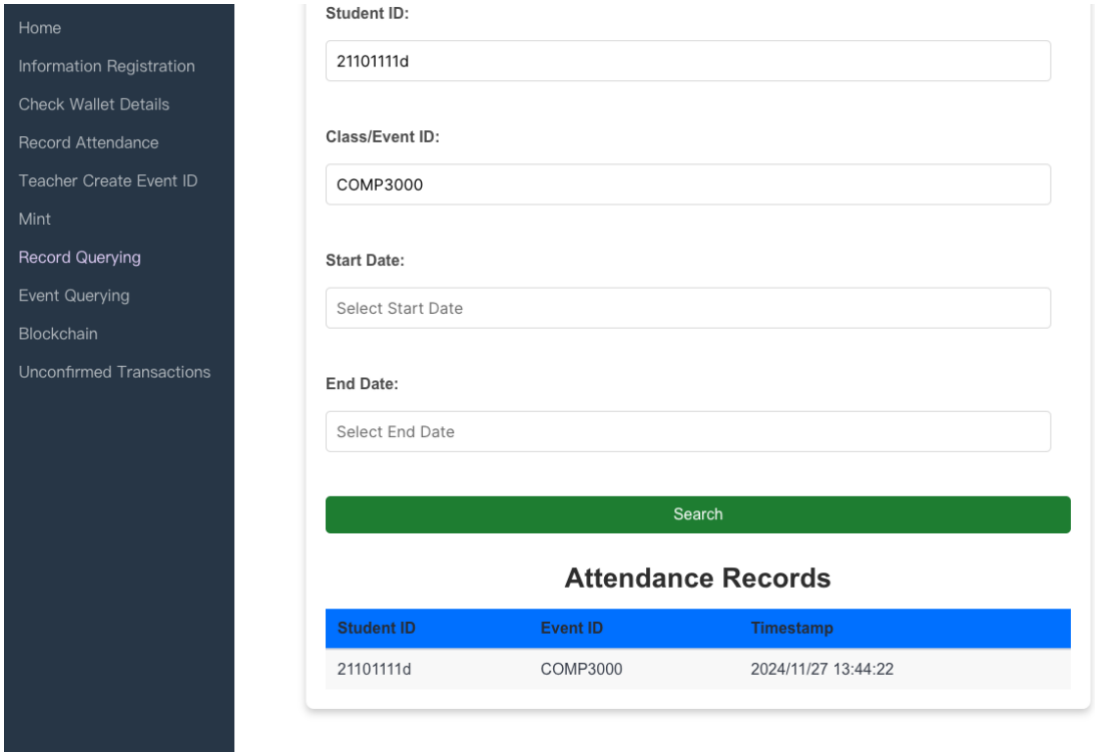
If we search a specific course, we can find the corresponding records:



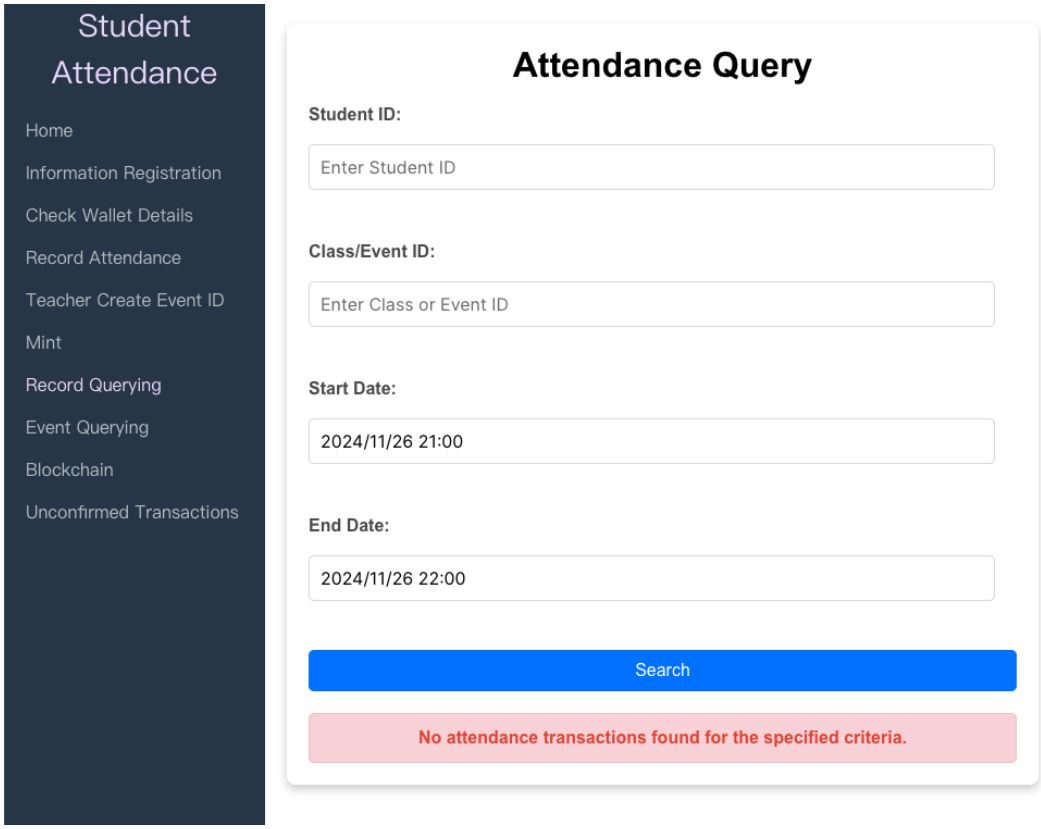
If we search a specific student, we can also find the corresponding records:



The queries can be combined:



If no record fits the condition, it will warn “No transaction found”:


Student Attendance
Home
Information Registration
Check Wallet Details
Record Attendance
Teacher Create Event ID
Mint
Record Querying
Event Querying
Blockchain
Unconfirmed Transactions
Attendance Query
Student ID:
Enter Student ID
Class/Event ID:
Enter Class or Event ID
Start Date:
2024/11/26 21:00
End Date:
2024/11/26 22:00
Search
No attendance transactions found for the specified criteria.